IN THE SPECIFICATION

Page 44, lines 11-15.

"Core libraries <u>10</u>, each providing a particular abstraction of the component.

Utility modules <u>12</u>, each performing an analysis that is supposed to be used in several contexts.

Mutability sub-analyses <u>14</u>, each serving to test one or more of the TestField conditions."

Page 45, lines 2-6.

"CFParse <u>16</u> provides information that reflects the structure of a particular classfile. The data-flow analysis requires some higher-level abstraction of control flow, either at the intra-procedural level (intra-procedural control flow graph) or at the inter-procedural level (call graph). On top of these abstractions two additional core libraries are implemented, each being an engine for data-flow analysis <u>18</u>:"

Page 46, line 6 to page 47, line 14.

"Type analysis <u>22</u>: used by both *TestField* routines as part of the tests for conditions B,D, E. For each analyzed method, the type analysis <u>22</u> identifies for each instruction and for each frame location, the set of possible types of the location. The analysis distinguishes between the cases where the exact run-time type of the referenced object is known, and the cases where the run-time type is known up to assignment compatibility. The analysis also takes into account the scoping issues. In particular, if a field may be modified from outside the analysis scope, its type is conservatively assumed to be known up to assignment compatibility with the declared type. The analysis <u>22</u> uses the intra-procedural engine.

Reachability analysis <u>24</u>: used only by the version of the *TestField* routine for static fields to test conditions B, D, E as a basis for other sub-analyses. For example, in order to determine whether a class variable's encapsulation is broken, the reachability analysis <u>24</u> is used to decide whether the object whose reference is being stored or returned may be reachable from a static field. Likewise, in order to determine whether a class variable's state is modified by a putfield instruction, the reachability analysis <u>24</u> is used to

decide whether the modified variable may be reachable from a static field. For each analyzed method, the reachability analysis 24 identified for each instruction and for each frame location referring to a mutable object, the set of escaping objects and class variables from which that object becomes reachable. The set of escaping objects includes objects reachable from the method parameters and the returned object. Note that reachability relationships of immutable objects are ignored. Mutability is determined based on a list of *immutable* types which is a parameter to the analysis.

The analysis 24 uses both the inter- and intra-procedural engines."

Page 47, line 16 to page 48, line 3.

"The next layer of the Mutability Analyzer (as described in FIG. 1) is the set of sub-analyses 14:

Value modification 26: used by both TestField routines to test for condition A. For each analyzed method, the value modification analysis 26 identifies the set of fields whose corresponding instance variables and class variables may be set within that method. Three cases are distinguished:

Page 49, line 1 to page 50, line 17.

"*Object modification* 28 : used by the static fields version of the *TestField* routine to test for condition B.

For each analyzed method, the state modification analysis 30 identifies the set of reference-type static fields and method parameters whose referenced object's state may be modified by this method. For each putfield or xastore instruction, the analysis consults the reachability analysis to determine whether the object being stored into is reachable from a static field or a method parameter. Note that the analysis makes no exception for initialization methods (‹init› or ‹clinit›); e.g., if a static field is first assigned a reference to an object during the corresponding class initialization method, and the state of this object is subsequently modified, the analysis would identify this as state modification.

The analysis uses the inter- and intra-procedural engines and the type and ·

reachability analyses.

Variable Accessibility 32: used by both versions of the *TestField* routine to test for condition C.

For each analyzed field, the variable accessibility analysis 32 identifies whether the value of the variable may be modified from outside the analysis scope. The current implementation is based only on the access restrictions defined by the language. Thus, a field which is non-`private` and non-`final` is identified as accessible. In other embodiments, this analysis can be improved by taking into account runtime access restrictions and additional scoping information.

Object Accessibility and *Breakage of Encapsulation* 34: used by the `static` fields version of the *TestField* routine to test for conditions D and E.

For each analyzed method, the encapsulation analysis 36 identifies the set of reference-type `static` fields and method parameters which may not be encapsulated upon the completion of the method. The definitions of encapsulation regard a variable as encapsulated unless there exists a mutable object reachable from the variable, which is accessible from outside the analysis scope. The implementation in the preferred embodiment is more conservative and regards creation of any non-local reference to a mutable object reachable from the variable as breakage of encapsulation of this variable. This is due to the difficulty in tracking non-local references."

Page 62, lines 11 to page 63, line 1.

"One of the core design decisions that drove the implementation of the preferred embodiment was to use basic core libraries such as CFParse 16 and JAN 20, and introduce general-purpose engines for intra-procedural and inter-procedural analyses. The code is designed to be scalable and fit into a multi-level static analysis framework, so that utilities and sub-analyses can be used and extended to deal with properties other than mutability characterization.

Static analysis is in some cases limited. Therefore, for properties that the analysis will

28

not be able to detect statically, smart annotations will be facilitated in other embodiments so as to detect those cases at run time. This can be done by using the CFParse 16 core library to parse, edit and annotate classfiles."